Qub: A Resource Aware Functional Programming Language

APOORV INGLE, The University of Kansas, USA

1 PROBLEM AND MOTIVATION

Managing resources—file handles, database connections, etc.—is a hard problem. Debugging resource leaks and runtime errors due to resource mis-management are difficult in evolving production code. Programming languages with static type systems are great tools to ensure erroneous code is detected at compile time. However, modern static type systems do little in the aspect of resource management as resources are treated as normal values. We propose a type system, **Qub**, based on the logic of bunched implications (***BI***)[14] which models resources as first class citizens. We distinguish two kinds of program objects—restricted and unrestricted—and two kinds of functions—sharing and separating. Our approach guarantees resource correctness without compromising existing functional abstractions.

For a concrete example, we consider the case of file handling. In Haskell, a file being closed twice or a file not being closed at all may cause run-time errors but it not flagged as a type error. We represent separating functions, i.e. functions that do not share resources with their arguments using $\multimap$, and sharing functions i.e. functions that share resources with their arguments using $\twoheadrightarrow$. In **Qub**, the type signatures of the file handling API explicitly states that they are separating in nature. This accounts for closing the file handle more than once. Each program object needs to be explicitly dropped if it has to be treated as a resource, as in linear type systems [1, 2, 11]. This accounts for failing to close the file handles.

Exception handling in Haskell can be done using `MonadError`[10]. However, it does not give a systematic way of cleaning up resources in case of run-time exceptions. We consider the case where a critical section of the code throws an exception as shown in Fig. 1. The `IOF` describes the fact that the computation can throw exceptions, while `IO` does not. The `catch` function has a sharing argument, hence it can access the file handle `fh` declared in the part of the code that can throw exceptions and close it before exiting to prevent a memory leak.

```
openFile :: FilePath ⊸ IO FileHandle
closeFile :: FileHandle ⊸ IO ()
readFile :: FileHandle
         ⊸ IOF (String, FileHandle)
writeFile :: String
          ⊸ FileHandle
          ⊸ IOF ((),   FileHandle)

throw :: Exception ⊸ IO a
catch :: IOF a ⊸ (Exception ⊸ IO a)
      ↠ IO a
```

```
readFromFile :: FilePath
             ⊸ IO (Either String String)
readFromFile fpath =
do fh ← openFile fpath
   (( s,  fh) ← readLine fh
   let l = caps s
   closeFile fh
   return $ Right l)
      `catch` (\e → do closeFile fh
                       return $
                         Left "read file error")
```

Fig. 1. File and Exception Handling in **Qub**

Advisor: J. Garrett Morris.
Author's address: Apoorv Ingle, ACM ID: 7456710, Information and Telecommunication Technology Center, The University of Kansas, USA, ani@ku.edu.

2 BACKGROUND AND RELATED WORK

Type systems based on linear logic[1–3, 11, 16] provide one technique to solve the resource control problem. They restrict the structural rules of weakening and contraction to view all values as resources. This changes the meaning of the connectives as well. Linear implication $A \multimap B$ means "A is consumed to obtain B". We also get additive and multiplicative fragments of conjunction ($A \otimes B$ means "both A and B" and $A \,\&\, B$ means "choose between A and B"). There is, however, an awkward asymmetry in this system—while $\multimap$ is the right adjoint of $\otimes$, & has no such counterpart. Logic of ***BI*** [15] repairs this asymmetry between implication and conjunction. It uses trees as contexts, where the internal nodes are either comma (,) or semicolon (;) and leaf nodes are the propositions. The structural rules—weakening and contraction—are prohibited for propositions connected using (,). $\Gamma; \Delta \vdash \Gamma$ but $\Gamma, \Delta \not\vdash \Gamma$. The multiplicative conjunction $\otimes$ gets a multiplicative implication $\mathrel{-\!\!*}$ and the additive conjunction & gets the additive implication $\twoheadrightarrow$ as its right adjoint. The Curry-Howard interpretation of ***BI*** is in terms of sharing in rather than linear logic's consumption. If the function does not share resources with its argument $\mathrel{-\!\!*}$ is used, while if the function shares resources with its arguments, $\twoheadrightarrow$ is used instead.

Jones[4, 8] introduces qualified types, a general framework to incorporate predicates for polymorphism. The Hindley-Milner type system[12] extended with qualified types[5] can express type classes with functional dependencies[7], and first class polymorphism[6]. Morris[13] uses qualified types to design Quill, a functional language with linear calculus. In Quill, the predicate $\mathsf{Un}\ \tau$ specifies the type τ is unrestricted i.e. it can be duplicated or dropped at will, or it does not contain any resources. Proof theoretically, the type is tagged unrestricted whenever weakening and contraction is admissible. A binary predicate $\geq$ helps generalize function definition in presence of restricted types. $\tau \geq \tau'$ specifies that type τ admits more structural rules than type τ'.

3 APPROACH AND UNIQUENESS

Qub is an extension of standard call-by-name lambda calculus based on logic of ***BI***. We introduce two kinds of lambdas associated with the two implications. $\lambda^{-\!*}x.M$ introduces a separating function $\mathrel{-\!\!*}$, while $\lambda^{\twoheadrightarrow}x.M$ introduces a sharing arrow $\twoheadrightarrow$. We generalize the use of trees as contexts in ***BI*** to graphs of sharing information. We represent sharing graphs as adjacency lists in the environment context. A triple $(x^{\vec{y}} : \tau) \in \Gamma$ would mean x of type τ is in sharing with $\vec{y}$. The sharing relation is a symmetric, reflexive and non-transitive. We say that the contexts are in complete sharing—$\Gamma \oplus \Delta$—if all the variables are shared and they are disjoint—$\Gamma \circledast \Delta$—if they are not shared. We formally define them in Fig. 2, where # means disjoint. The predicates $\mathsf{ShFun}\ \phi$ and $\mathsf{SeFun}\ \phi$ range over sharing and separating functions respectively. We include predicates $\mathsf{Un}\ \tau$ and $\tau \geq \tau'$ as is from Quill. The complete type system is shown in Fig. 3.

$$\mathsf{Vars}(\Gamma, x^{\vec{y}}) = \mathsf{Vars}(\Gamma) \cup \{x\} \qquad (\Gamma, x^{\vec{y}})^{[a \mapsto \vec{b}]} = \begin{cases} a \notin \vec{y} & (\Gamma^{[a \mapsto \vec{b}]}, x^{\vec{y}} : \tau) \\ a \in \vec{y} & (\Gamma^{[a \mapsto \vec{b}]}, x^{(\vec{y} \setminus a) \cup \vec{b}} : \tau) \end{cases}$$

$$\mathsf{Shared}(\Gamma, x^{\vec{y}}) = \mathsf{Shared}(\Gamma) \cup \{\vec{y}\}$$

$$\mathsf{Used}(\Gamma) = \mathsf{Vars}(\Gamma) \cup \mathsf{Shared}(\Gamma) \qquad \Gamma^{[\vec{a} \mapsto \vec{b}]} = (\ldots((\Gamma^{[a_1 \mapsto \vec{b}]})^{[a_2 \mapsto \vec{b}]})\cdots)^{[a_n \mapsto \vec{b}]}$$

$$\Gamma \circledast \Gamma' = \Gamma \sqcup \Gamma' \quad \textit{if}\ \mathsf{Vars}(\Gamma) \,\#\, \mathsf{Used}(\Gamma') \wedge \mathsf{Vars}(\Gamma') \,\#\, \mathsf{Used}(\Gamma)$$

$$\Gamma \oplus \Gamma' = \Gamma \sqcup \Gamma' \quad \textit{if}\ \mathsf{Used}(\Gamma) = \mathsf{Used}(\Gamma')$$

Fig. 2. Auxiliary Functions

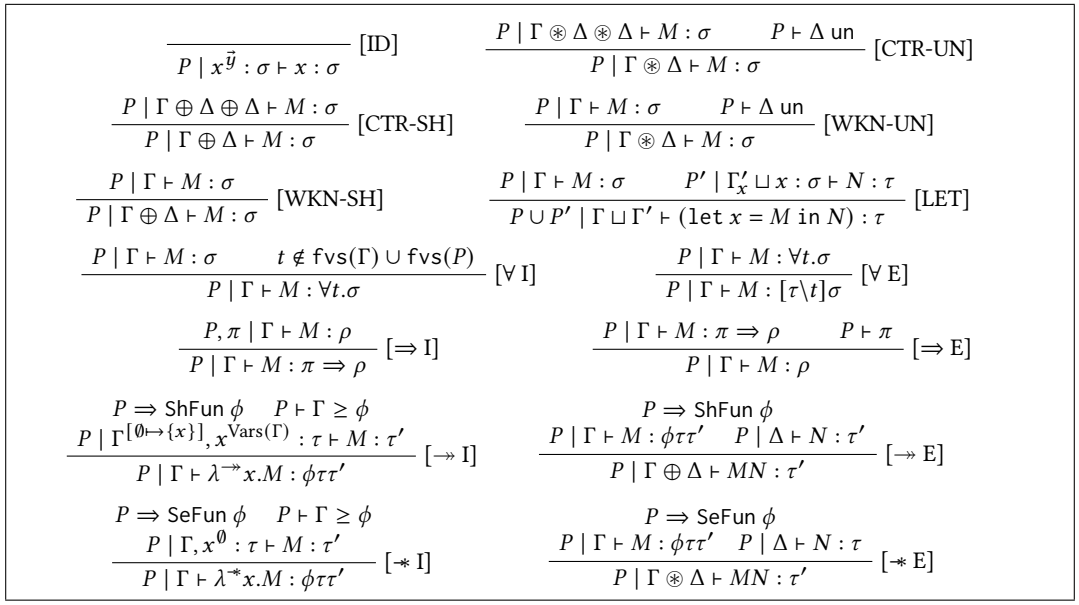

Fig. 3. **Qub** Type System

4 RESULTS AND CONTRIBUTIONS

Qub is a novel sub-structural λ-calculus that generalizes Curry-Howard Interpretation of ***BI***. We have developed a sound and complete syntax directed **Qub** type system and designed a type inference algorithm based on Algorithm $\mathcal{M}$[9]. We have extended our system to support kinds with user defined type constructors allowing programmers to define data types with sharing and separating fields. The use of monads with sharing and separating functions can statically detect resource errors, while expressing patterns like exceptions and non-determinism that are difficult to capture in linear languages as described in previous section.

REFERENCES

[1] Ahmed, A., Fluet, M., and Morrisett, G. L^3: A linear language with locations. *Fundamenta Informaticae 77*, 4 (2007), 397–449.

[2] Bernardy, J.-P., Boespflug, M., Newton, R. R., Peyton Jones, S., and Spiwack, A. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang. 2* (2017), 5:1–5:29.

[3] Girard, J.-Y. Linear logic. *Theoretical Computer Science 50*, 1 (1987), 1–101.

[4] Jones, M. P. A theory of qualified types. *Science of Computer Programming 22*, 3 (1994), 231 – 256.

[5] Jones, M. P. Simplifying and improving qualified types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (1995), FPCA '95, ACM, pp. 160–169.

[6] Jones, M. P. First-class polymorphism with type inference. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), POPL '97, ACM, pp. 483–496.

[7] Jones, M. P. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming* (2000), Springer-Verlag LNCS 1782.

[8] Jones, M. P. *Qualified Types: Theory and Practice.* Cambridge University Press, 2003.

[9] Lee, O., and Yi, K. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst. 20*, 4 (1998), 707–723.

[10] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), POPL '95, ACM, pp. 333–343.

[11] Mazurak, K., Zhao, J., and Zdancewic, S. Lightweight linear types in system F°. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (2010), TLDI '10, ACM, pp. 77–88.

[12] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348–375.

[13] Morris, J. G. The best of both worlds: Linear functional programming without compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (2016), ICFP 2016, ACM, pp. 448–461.

[14] O'Hearn, P. W., and Pym, D. J. The logic of bunched implications. *The Bulletin of Symbolic Logic 5*, 2 (Jan. 1999), 215–244.

[15] Pym, D. J. *The Semantics and Proof Theory of the Logic of Bunched Implications.* Applied Logic Series. Springer Netherlands, 2002.

[16] Wadler, P. A taste of linear logic. In *Mathematical Foundations of Computer Science 1993: 18th International Symposium, MFCS'93 Gdańsk, Poland, August 30–September 3, 1993 Proceedings*, A. M. Borzyszkowski and S. Sokolowski, Eds. Springer Berlin Heidelberg, 1993, pp. 185–210.